An Abstract Domain Combinator for Separately Conjoining Memory Abstractions

Antoine Toubhans, Bor-Yuh Evan Chang, Xavier Rival

An abstract domain combinator for separately conjoining memory abstractions ⋆

Antoine Toubhans[1], Bor-Yuh Evan Chang[2], and Xavier Rival[1]

[1] INRIA, ENS, CNRS, Paris, France
[2] University of Colorado, Boulder, Colorado, USA
toubhans@di.ens.fr, bec@cs.colorado.edu, rival@di.ens.fr

Abstract. The breadth and depth of heap properties that can be inferred by the union of today's shape analyses is quite astounding. Yet, achieving scalability while supporting a wide range of complex data structures in a generic way remains a long-standing challenge. In this paper, we propose a way to side-step this issue by defining a generic abstract domain combinator for combining memory abstractions on disjoint regions. In essence, our abstract domain construction is to the separating conjunction in separation logic as the reduced product construction is to classical, non-separating conjunction. This approach eases the design of the analysis as memory abstract domains can be re-used by applying our separating conjunction domain combinator. And more importantly, this combinator enables an analysis designer to easily create a combined domain that applies computationally-expensive abstract domains only where it is required.

1 Introduction

While there exist static analyses for the most common data structures such as lists, trees, or even overlaid lists and trees [10,4,14,11], it is uncommon for static analyses to efficiently support all of these simultaneously. For instance, consider the code fragment of Fig. 1 that simultaneously manipulates linked lists and trees, iteratively picking some value from the list and searching for it in the tree. Although verification of the memory safety and data structures preservation is possible with several tools (e.g. [18,20]), this will not take into account most efficient data-structure-specific algorithms (e.g. analysis for linked lists presented in [15] achieves polynomial complexity transfer functions). On the other hand, using only a linked-list-specific efficient analysis will lead to a dramatic loss of precision, as tree features are not supported. The general problem is much broader than just lists and trees: in real-world programs, it is common to find not only lists, trees, and overlaid lists and trees, but also buffers, arrays, and other complex heap structures, and therefore static analysis is either imprecise or inefficient.

Instead of using one monolithic analysis, we propose to combine off-the-shelf data-structure-specific analyses that reason about *disjoint* regions of memory. The approach presented in this paper is in the context of abstract interpretation [6]. Therefore, combining analyses is realized by a *separating combination* of memory abstract domains called

⋆ The research leading to these results has received funding from the European Research Council under the FP7 grant agreement 278673, Project MemCAD and the United States National Science Foundation under grant CCF-1055066.

```
01 : int *x = NULL;   10 : while(...){              24 :   r = searchTree(*x, t);
02 : List *h, *e;     11 :   x = malloc(sizeof(int));  25 :   // do something on r
03 : Tree *t, *r;     12 :   *x = e -> data             ...
  ...                   ...                         30 : }
```

Fig. 1. A code fragment manipulating simultaneously several data structures

the *sub-domains*. Combined memory abstract domains describe each disjoint memory region using one of its sub-domains. We show how *separation* (i.e. the fact that data structures do not share cell blocks) can be used to decompose a heterogeneous memory into several sub-instances that can be handled independently.

This construction increases and eases the abstract domain design capabilities. Combined abstract domains are more extensible and flexible, as a sub-domain can be individually added, removed, strengthened, or weakened in the combination. Moreover, it allows paying the cost of complex algorithms that usually come with expressive abstract domains only in the memory region that really requires it. On the other hand, simpler light-weight abstract domains can be used to represent a significant part of the memory that does not contain complex structures.

Such a combination poses several challenges. Because, even if disjoint in memory, concrete data structures can still be correlated (e.g. have shared values or pointers to each other), we need to carefully abstract the *interface* between memory regions in the combination. Maintaining a right partitioning (i.e. which memory region should be abstracted in which sub-domain) during the analysis process is also challenging. For example, when analyzing a memory allocation, the analysis decides which sub-domains should handle it. Even though any choice is sound, there are sub-domains more relevant than others in many cases. This approach has been successfully applied to numerical domains and made it possible to obtain scalable and precise analyses [3] and reusable abstract domains [13]. Our proposal brings, in a way, the same improvement to memory abstract domains. We justify this statement by the following contributions:

- we introduce (Section 2) and formalize (Section 3.2) the separating combination functor that takes two memory abstract sub-domains matching the signature given in Section 3.1 and returns a new combined memory abstract domain;
- we define an abstract domain for the interface between memory regions (Section 3.2) that carefully describes correlations between memory regions;
- we set up the abstract transfer functions (Section 4) that compose abstract transfer functions of sub-domains and extract information from an abstract interface;
- we give a heuristic for the decision of which sub-domain should handle a newly allocated block (Section 4.1);
- we evaluate the separating combination functor by an implementation in the MemCAD analyzer (Section 5) and empirically verify that combined analyses remain efficient and precise while offering greater flexibility.

2 Overview

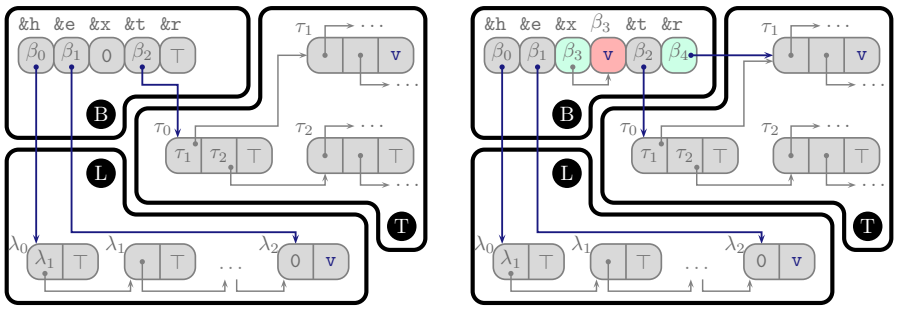
In this section, we provide an informal description of our combined analysis (formal details are presented in Sections 3 and 4). We present an abstract interpretation [6]

based analysis of the code fragment of Fig. 1, using a combination of memory abstract domains. The analysis goal is to prove memory safety and data structure (lists and trees) preservation. Fig. 2 shows two abstract memories computed during the analysis.

An abstraction of the memory using several memory abstract domains. The program manipulates a memory that can be decomposed in three disjoint regions (i) the list region (denoted L) containing linked-list nodes (ii) the tree region (denoted T) containing tree nodes (iii) the region accounting for the rest of the memory that contains only *bounded* data structures (denoted B). This naturally leads to the choice of a separating combination of three memory abstract sub-domains $\mathbb{M}_l^\sharp$, $\mathbb{M}_t^\sharp$, $\mathbb{M}_b^\sharp$ that will reason respectively about region L, T and B.

Fig. 2(a) shows the combined abstract memory computed by the analysis before line 11. Each thick black bordered boxes (labeled B, L and T) contains an element $m_b^\sharp \in \mathbb{M}_b^\sharp$, $m_l^\sharp \in \mathbb{M}_l^\sharp$ and $m_t^\sharp \in \mathbb{M}_t^\sharp$ called *abstract sub-memories*. Greek letters that appear in the sub-memories are called the *symbolic names* and are used by sub-domains to internally represent concrete values and heap addresses. The combined abstract memory represents a set of memories where variable `h` (resp. `e`) points to the head (resp. the last element) of a linked list, variable `t` points to the root of a tree, variable `x` is the null pointer and content of variable `r` can be any concrete value.

Describing the interface between memory regions is crucial for precision. In particular, the combined abstract memory should account for (i) pointers from region B to regions L and T and (ii) sharing of values between cells of different memory regions such as value `v` in the last list node and the left tree node from the tree root. The *interface abstract domain* $\mathbb{I}^\sharp$ (Section 3.2) achieves this by maintaining a set of equalities between symbolic names of different abstract sub-memories. An equality between two symbolic names simply means that they represent the *same* concrete value. For instance, pointer `h` crossing the memory regions B and L is represented by (i) β_0 representing the content of cell `h` in the $m_b^\sharp$, (ii) λ_0 representing the address of the head of the list in the $m_l^\sharp$, and (iii) equality $\beta_0 = \lambda_0$ in the abstract interface. Thus, this combined abstract memory is a quadruple made of three abstract sub-memories $m_b^\sharp \in \mathbb{M}_b^\sharp$, $m_l^\sharp \in \mathbb{M}_l^\sharp$ and $m_t^\sharp \in \mathbb{M}_t^\sharp$ and an abstract interface $i^\sharp \in \mathbb{I}^\sharp$. In the two combined abstract memories shown in Fig. 2, abstract sub-memories are represented in gray inside the thick black boxes whereas dark blue edges and values depict the abstract interface.



(a) Abstract memory state before line `11` (b) Abstract memory state after line `24`

Fig. 2. Two combined abstract memories inferred during the analysis

Combined analysis. The analysis automatically derives the post-condition shown in Fig. 2(b) from the pre-condition shown in Fig. 2(a) by composing abstract transfer functions for program statements between line 11 and 24. In the following, we demonstrate the key features of the combination of memory abstract domains by analyzing the two program statements at line 11 and 12. In particular, the assignment at line 11 involves a memory allocation but remains quite simple compared to the assignment at line 12 that requires post-condition computations to be distributed *across* sub-memories using the abstract interface. The first step when analyzing an assignment consists in *evaluating* its left and right hand-sides, that is, finding symbolic names representing the address of the updated cell and the written value (Section 4.2).

The right hand side of the assignment at line 11 contains a memory allocation instruction, hence the analysis should decide which sub-domain is the most relevant for handling it. Indeed, even though any choice is sound for the analysis, a bad decision could lead to a loss in precision or in efficiency. While the three sub-domains could precisely handle the memory allocation, $\mathbb{M}_b^\sharp$ is expected to be more efficient for handling *bounded* data structures. Because of the type passed to **malloc**, the new cell will likely never be *summarized* as part of a list or a tree, $m_b^\sharp$ should therefore handle the memory allocation. While not complete (C types cannot fully describe the programmer intended data structure), we find that this simple heuristic works well in practice. In Fig. 2(b), this sub-memory contains a new cell at abstract address β_3 (red background highlights created cells). The right hand side is evaluated to β_3 and evaluation of the left hand side is trivial and provides a symbolic name that is in $m_b^\sharp$ so that both are evaluated in the *same* sub-memory. Therefore and thanks to the local reasoning principle, the post-condition can be computed *only* for abstract sub-memory $m_b^\sharp$, using the sound abstract transfer function provided by $\mathbb{M}_b^\sharp$. In Fig. 2(b), the cell that correspond to `x` has been updated to β_3 (green background highlights updated cells).

Computing an abstract post-condition of the assignment at line 12 poses two main issues. First, the evaluation of its right hand side *crosses* abstract sub-memories that involves a mechanism for *extracting* information from $i^\sharp$. Secondly, the left and right hand sides are not evaluated in the same sub-memories. Thus, accurately handling this assignment requires sound *abstract transfer functions* for $\mathbb{I}^\sharp$. The right hand side is evaluated iteratively over its syntax: (i) the content of cell `e` is evaluated to β_1 in $m_b^\sharp$ (ii) β_1 cannot be resolved as an address of a cell in $m_b^\sharp$ (iii) equality $\beta_1 = \lambda_2$ can be *extracted* from the abstract interface $i^\sharp$, that allows the resolution to continue in $m_l^\sharp$ (iv) the content of abstract address λ_2 at field $\underline{\texttt{data}}$ is `v` in $m_l^\sharp$. As $*\texttt{x}$ is evaluated to β_3 in $m_b^\sharp$, the left and right hand sides are evaluated respectively in $m_b^\sharp$ and $m_l^\sharp$. Thus, the post-condition will be computed (i) in $m_b^\sharp$ as the written cell is abstracted in this abstract sub-memory (ii) in $i^\sharp$ as the written value is not represented in $m_b^\sharp$. The cell whose abstract address is β_3 is updated to a *fresh* symbolic name in $m_b^\sharp$, that is then set to be equal to `v` in $i^\sharp$ using the abstract transfer function for $\mathbb{I}^\sharp$.

3 The separating combination of memory abstract domains

In this section, we first set up a general notion of a *memory abstract domain* (Section 3.1). before introducing the *separating combination* (Section 3.2). A memory abstract domain $\mathbb{M}^\sharp$ provides a representation for sets of concrete memories. Intuitively, it consists of a set of predicates describing memory quantified on *symbolic names* (denoted $\mathcal{N}_{\mathbb{M}^\sharp}$) that represent *concrete values* (denoted $\mathbb{V}$). Thus, concretization involves *valuations* mapping symbolic names to the value they represent. Once this general notion is formalized, we formally introduce the separating combination as a binary functor that takes as input two memory abstract domains $\mathbb{M}_1^\sharp$, $\mathbb{M}_2^\sharp$ and returns a new memory abstract domain $\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp$. The functor can be iteratively applied in order to cope with more than two memory abstract sub-domains. The combined abstract domain describes disjoint memory regions using either predicates of $\mathbb{M}_1^\sharp$ or $\mathbb{M}_2^\sharp$. Moreover, correlations between regions are described by the *interface abstract domain* $\mathbb{I}^\sharp$ that maintains equalities between symbolic names quantified in different sub-memories.

3.1 Memory abstract domain

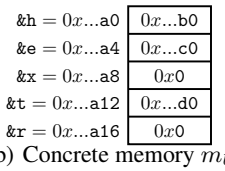
Concrete memories. We let $\mathbb{A}$ denote the set of concrete addresses, and we assume addresses to be concrete values (i.e. $\mathbb{A} \subseteq \mathbb{V}$). Henceforth, we adopt a standard model for concrete memories where a concrete memory m is a finite map from addresses to values. Therefore, the set of concrete memories is defined by $\mathbb{M} \stackrel{\text{def}}{=} \mathbb{A} \rightharpoonup_{\text{fin}} \mathbb{V}$. We let $\mathbb{F} = \{\underline{\mathrm{f}}, \underline{\mathrm{g}}, \ldots\}$ denote the set of valid field names, and we treat them as numerical offsets so that for $a \in \mathbb{A}$ and $\underline{\mathrm{f}} \in \mathbb{F}$, $a + \underline{\mathrm{f}}$ denotes the address at field $\underline{\mathrm{f}}$ of the block at address a. As the separating combination is reasoning about disjoint memory region, we write $m_1 \uplus m_2$ for the *union* of two *disjoint* memories (i.e. $\mathbf{dom}(m_1) \cap \mathbf{dom}(m_2) = \emptyset$, where $\mathbf{dom}(m_i)$ denotes the domain of m_i as a partial function).

A *memory abstract domain* is a lattice of abstract memories $\mathbb{M}^\sharp$, together with a fixed infinite set of symbolic names $\mathcal{N}_{\mathbb{M}^\sharp}$, a concretization function $\gamma_{\mathbb{M}^\sharp}$ and sound abstract transfer functions (detailed in Section 4). An abstract memory $m^\sharp \in \mathbb{M}^\sharp$ internally utilizes symbolic names to represent concrete values. We define the set of valuations $\mathcal{V}_{\mathbb{M}^\sharp} \stackrel{\text{def}}{=} \mathcal{N}_{\mathbb{M}^\sharp} \rightharpoonup_{\text{fin}} \mathbb{V}$. Intuitively, a valuation $\nu \in \mathcal{V}_{\mathbb{M}^\sharp}$ relates symbolic names to concrete values when concretizing. Concretization is a function $\gamma_{\mathbb{M}^\sharp} : \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{M} \times \mathcal{V}_{\mathbb{M}^\sharp})$ and $\gamma_{\mathbb{M}^\sharp}(m^\sharp)$ collects a set of couples $(m, \nu) \in \mathbb{M} \times \mathcal{V}_{\mathbb{M}^\sharp}$ made of a concrete memory and a valuation that maps symbolic names quantified in $m^\sharp$ to concrete values in m.

Example 1 (Bounded structure abstract domain). As a first example, we describe a memory abstract domain that represents precisely block contents, but is unable to summarize unbounded regions such as list and tree data structures. This memory abstract domain can be considered an instantiation of $\mathbb{M}_b^\sharp$ seen in the overview (Section 2). The set of symbolic names consists of either symbols for addresses (denoted $\alpha_0^a, \alpha_1^a, \ldots$) or symbols for cell contents (denoted $\beta_0^c, \beta_1^c, \ldots$). Definitions of abstract memories and

$$m_b^\sharp = \alpha_0^a \mapsto \{\underline{0} : \beta_0^c\} \ * \ \alpha_1^a \mapsto \{\underline{0} : \beta_1^c\} \ * \ \alpha_2^a \mapsto \{\underline{0} : \beta_2^c\} \ * \ \alpha_3^a \mapsto \{\underline{0} : \beta_3^c\} \ * \ \alpha_4^a \mapsto \{\underline{0} : \beta_4^c\}$$
$$\wedge \ \&\mathtt{h} = \alpha_0^a \ \wedge \ \&\mathtt{e} = \alpha_1^a \ \wedge \ \&\mathtt{x} = \alpha_2^a \ \wedge \ \&\mathtt{t} = \alpha_3^a \ \wedge \ \&\mathtt{r} = \alpha_4^a \ \wedge \ \beta_2^c = \mathtt{0}$$

(a) Abstract memory $m_b^\sharp$

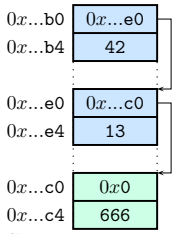


(b) Concrete memory m_b

ν_b : $\alpha_0^a \mapsto 0x...\mathtt{a0}$ $\quad \beta_0^c \mapsto 0x...\mathtt{b0}$
$\alpha_1^a \mapsto 0x...\mathtt{a4}$ $\quad \beta_1^c \mapsto 0x...\mathtt{c0}$
$\alpha_2^a \mapsto 0x...\mathtt{a8}$ $\quad \beta_2^c \mapsto 0x\mathtt{0}$
$\alpha_3^a \mapsto 0x...\mathtt{a12}$ $\quad \beta_3^c \mapsto 0x...\mathtt{d0}$
$\alpha_4^a \mapsto 0x...\mathtt{a16}$ $\quad \beta_4^c \mapsto 0x\mathtt{0}$

(c) Valuation $\nu_b : \mathcal{N}_{\mathbb{M}_b^\sharp} \to \mathbb{V}$

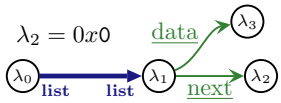
Fig. 3. Bounded data structure abstract domain: $(m_b, \nu_b) \in \gamma_{\mathbb{M}_b^\sharp}(m_b^\sharp)$



(a) Concrete memory m_l

ν_l : $\lambda_0 \mapsto 0x...\mathtt{b0}$ $\quad \lambda_2 \mapsto 0x\mathtt{0}$
$\lambda_1 \mapsto 0x...\mathtt{c0}$ $\quad \lambda_3 \mapsto \mathtt{666}$

(b) Valuation $\nu_l : \mathcal{N}_{\mathbb{M}_l^\sharp} \to \mathbb{V}$



(c) Abstract memory $m_l^\sharp$

Fig. 4. Separating shape graph abstract domain, parameterized by list inductive definition: $(m_l, \nu_l) \in \gamma_{\mathbb{M}_l^\sharp}(m_l^\sharp)$

the concretization function is given by:

$m^\sharp$ ::=	abstract memories	$\gamma_{\mathbb{M}_b^\sharp}(m^\sharp)$
$\mid \mathbf{emp}$	empty memory	$\{([\,], \nu) \mid \nu \in \mathcal{V}_{\mathbb{M}^\sharp}\}$
$\mid \alpha^a \mapsto \{\underline{\mathtt{f}} : \beta^c\}$	memory cell	$\{([\nu(\alpha^a) + \underline{\mathtt{f}} \mapsto \nu(\beta^c)], \nu) \mid \nu \in \mathcal{V}_{\mathbb{M}^\sharp}\}$
$\mid m_1^\sharp * m_2^\sharp$	disjoint memory	$\{(m_1 \uplus m_2, \nu) \mid \forall i \in \{1,2\}.\ (m_i, \nu) \in \gamma_{\mathbb{M}_b^\sharp}(m_i^\sharp)\}$
$\mid m_b^\sharp \wedge n^\sharp$	with constraints	$\{(m, \nu) \mid (m, \nu) \in \gamma_{\mathbb{M}_b^\sharp}(m_b^\sharp) \ \wedge \ \nu \vDash n^\sharp\}$

An abstract memory $m^\sharp$ consists of a separating conjunction of atomic predicates $\alpha^a \mapsto \{\underline{\mathtt{f}} : \beta^c\}$ abstracting a cell at address $\alpha^a + \underline{\mathtt{f}}$ of content β^c. Fig. 3 shows the abstract sub-memory $m_b^\sharp$ depicted in labeled box Ⓑ in Fig. 2(a) and a pair (m_b, ν_b) that concretizes it. Properties about values and addresses are expressed in $n^\sharp$, using a product with a numerical domain [5]. For instance, a numerical domain enabling linear equalities is used in Fig. 3(a). Besides, a product with a pointer domain may be used to capture, for example, aliasing relations. The memory abstract domain of [16] extends this basic layout (and handles unions, non-fixed cell sizes, etc.).

Example 2 (Separating shape graphs). The separating shape graph abstract domain of [4] provides a second example of a memory abstract domain. An abstract memory is a separating conjunction [17] of predicates, which could be either points-to predicates (depicted as thin edges in Fig. 4(c)) and inductive predicates (depicted as bold edges

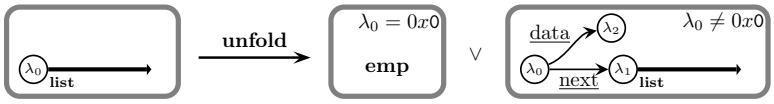


Fig. 5. List inductive definition rewriting rule

in Fig. 4(c)). Inductive predicates are annotated with inductive definitions supplied as a parameter of the domain. Thus, depending on the parameterization, this domain may provide an instantiation for $\mathbb{M}_l^\sharp$ or $\mathbb{M}_t^\sharp$ in the example of Section 2. Non-parameterizable abstract domains [10] have a similar layout. A graph containing only points-to edges is concretized into the disjoint merge of the cells described by each points-to edge. The concretization of inductive predicates proceeds by unfolding. For instance the inductive definition for list leads to the unfolding rule shown in Figure 5. As in the previous example, a numerical abstract domain should be used in order to express content properties. Fig. 4(c) presents an instance $m_l^\sharp$ of the separating shape graph domain parameterized by a list definition that corresponds to the labeled box ❶ depicted in Fig. 2(a) in the overview. A pair (m_l, ν_l) that concretizes $m_l^\sharp$ is given in Fig. 4(a) and Fig. 4(b).

3.2 The separating combination

In this section, we assume a pair of memory abstract domains $\mathbb{M}_1^\sharp, \mathbb{M}_2^\sharp$ are fixed, with independent sets of symbolic names $\mathcal{N}_{\mathbb{M}_1^\sharp}, \mathcal{N}_{\mathbb{M}_2^\sharp}$ and concretization functions $\gamma_{\mathbb{M}_1^\sharp}, \gamma_{\mathbb{M}_2^\sharp}$. In the following, we introduce the interface abstract domain before defining the combined memory abstract domain $\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp$.

Interface abstract domain. We let $\mathbb{I}^\sharp\langle\mathbb{M}_1^\sharp, \mathbb{M}_2^\sharp\rangle$ denote the interface abstract domain that expresses sets of equality relations between symbolic names of $\mathbb{M}_1^\sharp$ and $\mathbb{M}_2^\sharp$. Intuitively, an abstract interface is a finite set of pairs representing equalities. Thus, the interface abstract domain is defined by $\mathbb{I}^\sharp\langle\mathbb{M}_1^\sharp, \mathbb{M}_2^\sharp\rangle \stackrel{\text{def}}{=} \mathcal{P}_{\text{fin}}(\mathcal{N}_{\mathbb{M}_1^\sharp} \times \mathcal{N}_{\mathbb{M}_2^\sharp})$ and an abstract interface $i^\sharp$ is concretized into a set of pairs of valuations of $\mathbb{M}_1^\sharp$ and $\mathbb{M}_2^\sharp$ in the following way:

$$\gamma_{\mathbb{I}^\sharp}(i^\sharp) \stackrel{\text{def}}{=} \left\{(\nu_1, \nu_2) \in \mathcal{V}_{\mathbb{M}_1^\sharp} \times \mathcal{V}_{\mathbb{M}_2^\sharp} \mid \forall(\alpha_1, \alpha_2) \in i^\sharp.\ \nu_1(\alpha_1) = \nu_2(\alpha_2)\right\}$$

We write $\mathbb{I}^\sharp$ instead of $\mathbb{I}^\sharp\langle\mathbb{M}_1^\sharp, \mathbb{M}_2^\sharp\rangle$ when there is no ambiguity about the choice of the memory abstract sub-domains. We also define a judgment $i^\sharp \vdash \alpha_1 = \alpha_2$ meaning that the pair (α_1, α_2) belongs to the transitive closure of the relation induced by $i^\sharp$. Thus, it meets the soundness condition $i^\sharp \vdash \alpha_1 = \alpha_2 \wedge (\nu_1, \nu_2) \in \gamma_{\mathbb{I}^\sharp}(i^\sharp) \Rightarrow \nu_1(\alpha_1) = \nu_2(\alpha_2)$.

The separating abstract domain combinator. Combined abstract memories consist of triples $(m_1^\sharp, m_2^\sharp, i^\sharp)$ made of two abstract sub-memories describing disjoint memory regions and an abstract interface representing correlations between the sub-memories. Thus, the combined abstract domain is defined by $\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp \stackrel{\text{def}}{=} \mathbb{M}_1^\sharp \times \mathbb{M}_2^\sharp \times \mathbb{I}^\sharp\langle\mathbb{M}_1^\sharp, \mathbb{M}_2^\sharp\rangle$. We define the set of symbolic names of the combined abstract domain as the *disjoint*

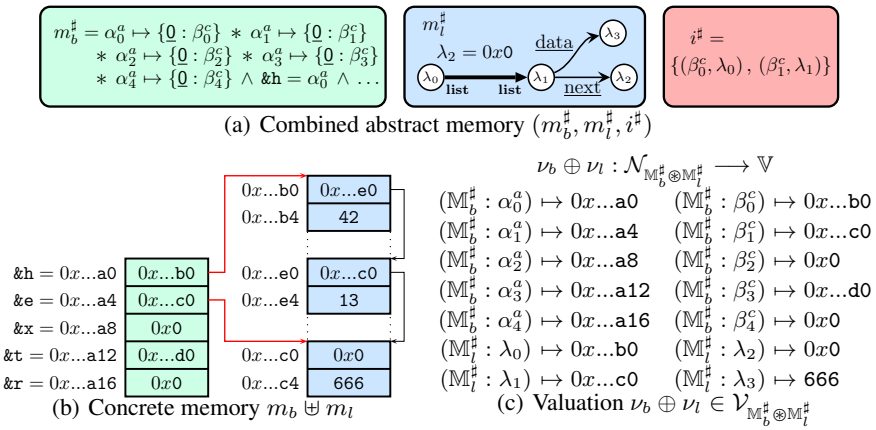


Fig. 6. Combined memory abstract domain: $(m_b \uplus m_l, \nu_b \oplus \nu_l) \in \gamma_{\mathbb{M}_b^\sharp \circledast \mathbb{M}_l^\sharp}(m_b^\sharp, m_l^\sharp, i^\sharp)$ as $(m_b, \nu_b) \in \gamma_{\mathbb{M}_b^\sharp}(m_b^\sharp)$ (Fig. 3), $(m_l, \nu_l) \in \gamma_{\mathbb{M}_l^\sharp}(m_l^\sharp)$ (Fig. 4) and $(\nu_b, \nu_l) \in \gamma_{\mathbb{I}^\sharp}(i^\sharp)$

union of symbolic names of the abstract sub-domains. Formally:

$$\mathcal{N}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp} \stackrel{\text{def}}{=} \left\{ (\mathbb{M}_1^\sharp : \alpha_1) \mid \alpha_1 \in \mathcal{N}_{\mathbb{M}_1^\sharp} \right\} \uplus \left\{ (\mathbb{M}_2^\sharp : \alpha_2) \mid \alpha_2 \in \mathcal{N}_{\mathbb{M}_2^\sharp} \right\}$$

At the combined abstract domain level, $(\mathbb{M}_i^\sharp : \alpha_i)$ denotes symbolic name α_i of the abstract sub-domain $\mathbb{M}_i^\sharp$. To define the meaning of a combined abstract memory, we give a concretization function $\gamma_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp} : \mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp \to \mathcal{P}(\mathbb{M} \times \mathcal{V}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp})$ that derives from concretization functions $\gamma_{\mathbb{M}_i^\sharp} : \mathbb{M}_i^\sharp \to \mathcal{P}(\mathbb{M} \times \mathcal{V}_{\mathbb{M}_i^\sharp})$ of the memory abstract sub-domains. To achieve this, we define a valuation *combinator* $\oplus : \mathcal{V}_{\mathbb{M}_1^\sharp} \times \mathcal{V}_{\mathbb{M}_2^\sharp} \to \mathcal{V}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}$ that puts two valuations aside:

$$(\nu_1 \oplus \nu_2)(\mathbb{M}_1^\sharp : \alpha_1) \stackrel{\text{def}}{=} \nu_1(\alpha_1) \qquad (\nu_1 \oplus \nu_2)(\mathbb{M}_2^\sharp : \alpha_2) \stackrel{\text{def}}{=} \nu_2(\alpha_2)$$

Then, the concretization of a combined abstract memory is given by:

$$\gamma_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}(m_1^\sharp, m_2^\sharp, i^\sharp) \stackrel{\text{def}}{=} \left\{ (m_1 \uplus m_2, \nu_1 \oplus \nu_2) \;\middle|\; \begin{array}{c} \forall i \in \{1,2\}.\, (m_i, \nu_i) \in \gamma_{\mathbb{M}_i^\sharp}(m_i^\sharp) \\ \wedge \quad (\nu_1, \nu_2) \in \gamma_{\mathbb{I}^\sharp}(i^\sharp) \end{array} \right\}$$

Example 3 (Separating combination of $\mathbb{M}_b^\sharp$ and $\mathbb{M}_l^\sharp$). We now consider an instantiation of the separating combination functor, with the bounded data structure domain $\mathbb{M}_b^\sharp$ (presented in Example 1) and the list-parameterized separating shape graph domain $\mathbb{M}_l^\sharp$ (presented in Example 2). Fig. 6(a) presents a combined abstract memory $(m_b^\sharp, m_l^\sharp, i^\sharp)$ that combines abstract sub-memories already presented in Fig. 3(a) and Fig. 4(c) together with the abstract interface $i^\sharp = \{(\beta_0^c, \lambda_0), (\beta_1^c, \lambda_1)\}$. We provide a pair (m, ν) in Fig. 6(b) and Fig. 6(c) concretizing $(m_b^\sharp, m_l^\sharp, i^\sharp)$ obtained by combining the concrete pairs (m_b, ν_b) and (m_l, ν_l) presented in Fig. 3 and Fig. 4. Note that $(\nu_b, \nu_l) \in \gamma_{\mathbb{I}^\sharp}(i^\sharp)$ as $\nu_b(\beta_0^c) = \nu_l(\lambda_0) = 0x...\texttt{b0}$ and $\nu_b(\beta_1^c) = \nu_l(\lambda_1) = 0x...\texttt{c0}$.

4 Analysis algorithms

We now discuss the inference of invariants in the combined domain. A memory abstract domain $\mathbb{M}^\sharp$ provides for each concrete memory operation $f : \mathbb{M} \to \mathcal{P}(\mathbb{M})$, a counterpart abstract transfer function $f^\sharp : \mathbb{M}^\sharp \to \mathbb{M}^\sharp$ that is *sound* (i.e. $\forall (m, \nu) \in \gamma_{\mathbb{M}^\sharp}(m^\sharp).\ \forall m' \in f(m).\ \exists \nu' \supseteq \nu.\ (m', \nu') \in (\gamma_{\mathbb{M}^\sharp} \circ f^\sharp)(m^\sharp)$). Abstract transfer functions may introduce new symbolic names but may not remove nor change the meaning of existing symbolic names. Hence, a valuation ν' in the concretization of the post-condition must extend valuation ν that concretizes the pre-condition. Abstract interpreters also require lattice operations (e.g. inclusion checking, widening) to achieve precise fixed point computations.

In a combined abstract domain, abstract transfer functions should distribute computations to the sub-memories and the abstract interface, using abstract transfer functions provided by sub-domains. In this section, we detail this mechanism for abstract transfer functions handling memory allocations (Section 4.1), assignments (Sections 4.2, and 4.3), and for inclusion checking (Section 4.4).

4.1 Creation of memory cells

Creation of new memory cells occurs either when a block for a new variable is created or when heap space is allocated at run time (e.g. **malloc** as at line 11 in Fig. 1). In a memory abstract domain $\mathbb{M}^\sharp$, this operation is handled by the abstract transfer function $\mathit{new}_{\mathbb{M}^\sharp}$, which is the abstract counterpart of the concrete transfer function $\mathit{new} : \mathtt{int} \times \mathbb{M} \to \mathcal{P}(\mathbb{A} \times \mathbb{M})$ (defined in a standard way). Intuitively, $\mathit{new}_{\mathbb{M}^\sharp}$ takes as input an integer size $\mathtt{s}$ and an abstract memory $m^\sharp_{\text{pre}}$ and returns a pair consisting of a symbolic name α representing the address of the allocated block and an abstract memory $m^\sharp_{\text{post}}$ where the cell has been created. Therefore, it ensures that, if $\mathit{new}_{\mathbb{M}^\sharp}(\mathtt{s}, m^\sharp_{\text{pre}}) = (\alpha, m^\sharp_{\text{post}})$ and $(m, \nu) \in \gamma_{\mathbb{M}^\sharp}(m^\sharp_{\text{pre}})$, then the following holds:

$$(a, m') \in \mathit{new}(\mathtt{s}, m) \Rightarrow \exists \nu' \supseteq \nu.\ (m', \nu') \in \gamma_{\mathbb{M}^\sharp}(m^\sharp_{\text{post}}) \ \wedge\ \nu'(\alpha) = a$$

Creation of memory cells in a combined domain. Because of the separation principle (Section 3.2), a cell must be represented in exactly one sub-memory in a combined abstract memory $m^\sharp = (m^\sharp_1, m^\sharp_2, i^\sharp)$. Therefore, we provide two possible definitions for $\mathit{new}_{\mathbb{M}^\sharp_1 \circledast \mathbb{M}^\sharp_2}$ deriving from two symmetric rules NEW1 and NEW2 (NEW2 is shown in Appendix A.1). Intuitively, the abstract transfer function defined by rule NEW1 (resp. NEW2) always represents new cells using sub-domain $\mathbb{M}^\sharp_1$ (resp. $\mathbb{M}^\sharp_2$).

$$\text{NEW1}\quad \frac{\mathit{new}_{\mathbb{M}^\sharp_1}(\mathtt{s}, m^\sharp_1) = (\alpha_1, m^\sharp_{1,\text{post}})}{\mathit{new}_{\mathbb{M}^\sharp_1 \circledast \mathbb{M}^\sharp_2}(\mathtt{s}, (m^\sharp_1, m^\sharp_2, i^\sharp)) = ((\mathbb{M}^\sharp_1 : \alpha_1), (m^\sharp_{1,\text{post}}, m^\sharp_2, i^\sharp))}$$

While both choices are *sound*, some sub-domains are more suitable than others. For instance, in $\mathbb{M}^\sharp_b \circledast \mathbb{M}^\sharp_l$, it would be inappropriate to let the allocation of a cell expected to be summarized as part of a list be done in $\mathbb{M}^\sharp_b$, where summarization cannot be achieved.

$$
\begin{array}{lll}
l\ (\in \mathcal{L}_{\mathbb{M}^\sharp}) ::= \alpha \quad (\alpha \in \mathcal{N}_{\mathbb{M}^\sharp}) & & r\ (\in \mathcal{R}_{\mathbb{M}^\sharp}) ::= l \quad (l \in \mathcal{L}_{\mathbb{M}^\sharp}) \\
\quad\quad | \quad l \cdot \underline{\mathrm{f}}\ (l \in \mathcal{L}_{\mathbb{M}^\sharp}; \underline{\mathrm{f}} \in \mathbb{F}) & & \quad\quad | \quad \&l\ (l \in \mathcal{L}_{\mathbb{M}^\sharp}) \\
\quad\quad | \quad *r \quad (r \in \mathcal{R}_{\mathbb{M}^\sharp}) & & \quad\quad | \quad v \quad (v \in \mathbb{V})
\end{array}
$$

(a) Syntax of l-value and r-value expressions

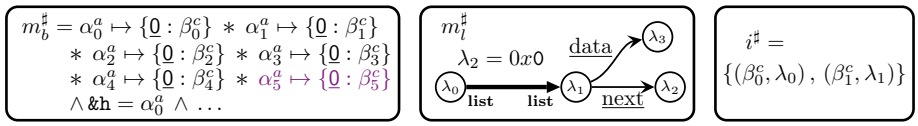
$$
\frac{}{\mathcal{L}_{\mathbb{M}^\sharp}[\![\alpha]\!]\, m^\sharp = (\alpha, \underline{0})} \qquad
\frac{\mathcal{L}_{\mathbb{M}^\sharp}[\![l]\!]\, m^\sharp = (\alpha, \underline{\mathrm{f}})}{\mathcal{L}_{\mathbb{M}^\sharp}[\![l \cdot \underline{\mathrm{g}}]\!]\, m^\sharp = (\alpha, \underline{\mathrm{f}} + \underline{\mathrm{g}})} \qquad
\frac{\mathcal{R}_{\mathbb{M}^\sharp}[\![r]\!]\, m^\sharp = \alpha + \underline{\mathrm{f}}}{\mathcal{L}_{\mathbb{M}^\sharp}[\![*r]\!]\, m^\sharp = (\alpha, \underline{\mathrm{f}})}
$$

$$
\frac{\mathcal{L}_{\mathbb{M}^\sharp}[\![l]\!]\, m^\sharp = (\alpha, \underline{\mathrm{f}})}{\mathcal{R}_{\mathbb{M}^\sharp}[\![l]\!]\, m^\sharp = \mathit{read}_{\mathbb{M}^\sharp}(\alpha, \underline{\mathrm{f}}, m^\sharp)} \qquad
\frac{\mathcal{L}_{\mathbb{M}^\sharp}[\![l]\!]\, m^\sharp = (\alpha, \underline{\mathrm{f}})}{\mathcal{R}_{\mathbb{M}^\sharp}[\![\&l]\!]\, m^\sharp = \alpha + \underline{\mathrm{f}}} \qquad
\frac{}{\mathcal{R}_{\mathbb{M}^\sharp}[\![v]\!]\, m^\sharp = v}
$$

(b) Evaluation rules for l-value and r-value expressions

Fig. 7. Evaluations of l-values $\mathcal{L}_{\mathbb{M}^\sharp}[\![l]\!] : \mathbb{M}^\sharp \to \mathcal{N}_{\mathbb{M}^\sharp} \times \mathbb{F}$ and r-values $\mathcal{R}_{\mathbb{M}^\sharp}[\![r]\!] : \mathbb{M}^\sharp \to \mathcal{E}_{\mathbb{M}^\sharp}$ only rely on the reading operation $\mathit{read}_{\mathbb{M}^\sharp} : \mathcal{N}_{\mathbb{M}^\sharp} \times \mathbb{F} \times \mathbb{M}^\sharp \to \mathcal{E}_{\mathbb{M}^\sharp}$

If we consider the analysis of the memory allocation at line 11 in Fig. 1, the choice is guided by C types: the created cell has type `int` which is not recursive, and thus it will likely never require summarization. Therefore, the cell creation can be handled by any sub-memories without any loss in precision. As sub-domain $\mathbb{M}_b^\sharp$ is more light-weight than $\mathbb{M}_l^\sharp$ in terms of computational cost, it should abstract the new cell. Then, invoking $\mathit{new}_{\mathbb{M}_b^\sharp \circledast \mathbb{M}_l^\sharp}$ deriving from rule NEW1 to the combined memory of Fig. 6(a) returns symbolic name $(\mathbb{M}_b^\sharp : \alpha_5^a)$ and the following combined abstract memory:



While not being critical for soundness, such empirical hints are important to avoid either a loss of precision or a slowdown in the analysis.

4.2 Evaluation of l-value and r-value expressions

We consider the abstract transfer functions handling operations such as assignments and tests. These operations involve *l-values* $l \in \mathcal{L}_{\mathbb{M}^\sharp}$ and *r-values* $r \in \mathcal{R}_{\mathbb{M}^\sharp}$. Their syntax (shown in Fig. 7(a)) includes classical forms of expressions encountered in a C-like language (structure fields, dereferences, address of, etc.). In this section, we define a mechanism for evaluating l-value and r-value expressions. More formally, the evaluation of an l-value l in abstract memory $m^\sharp$ returns a pair $\mathcal{L}_{\mathbb{M}^\sharp}[\![l]\!]\, m^\sharp = (\alpha, \underline{\mathrm{f}})$ consisting of a symbolic name α and a field $\underline{\mathrm{f}}$ such that $\alpha + \underline{\mathrm{f}}$ denotes the address represented by l. Similarly, the evaluation of a r-value r returns an *symbolic expression* $\mathcal{R}_{\mathbb{M}^\sharp}[\![r]\!]\, m^\sharp = e$ that denotes the value represented by r. A symbolic expression $e \in \mathcal{E}_{\mathbb{M}^\sharp}$ is either of the form $\alpha + \underline{\mathrm{f}}$ (where $\alpha \in \mathcal{N}_{\mathbb{M}^\sharp}$ and $\underline{\mathrm{f}} \in \mathbb{F}$) or a concrete value $v \in \mathbb{V}$.

Evaluation algorithm. The computation of $\mathcal{L}_{\mathbb{M}^\sharp}[\![.]\!]$ and $\mathcal{R}_{\mathbb{M}^\sharp}[\![.]\!]$ proceeds by induction over the expressions syntax as shown in Fig. 7(b), assuming a *read* operation $\mathit{read}_{\mathbb{M}^\sharp}$ is provided by memory abstract domain $\mathbb{M}^\sharp$, so as to "extract" the contents of a cell at the abstract level: partial function $\mathit{read}_{\mathbb{M}^\sharp}$ inputs a symbolic name α representing the base

address of a concrete block, a field $\underline{\mathrm{f}}$ and an abstract memory state $m^\sharp$, and returns a symbolic expression representing the contents of that field. It may also fail to identify the cell and is then undefined (this may happen in a combined memory abstract domain when reading a cell in the "wrong" sub-memory). In some memory abstract domains (such as the separating shape graph domain presented in Example 2), $\mathit{read}_{\mathbb{M}^\sharp}$ may need to perform *unfolding* [4] and thus, return a finite set of disjuncts, however this issue is orthogonal to the present development, so we leave it out here. Overall, it should satisfy the following soundness condition:

$$(m,\nu) \in \gamma_{\mathbb{M}^\sharp}(m^\sharp) \wedge \mathit{read}_{\mathbb{M}^\sharp}(\alpha, \underline{\mathrm{f}}, m^\sharp) = \beta + \underline{\mathrm{g}} \implies m(\,\nu(\alpha) + \underline{\mathrm{f}}\,) = \nu(\beta) + \underline{\mathrm{g}}$$
$$(m,\nu) \in \gamma_{\mathbb{M}^\sharp}(m^\sharp) \wedge \quad \mathit{read}_{\mathbb{M}^\sharp}(\alpha, \underline{\mathrm{f}}, m^\sharp) = v \quad \implies \quad m(\,\nu(\alpha) + \underline{\mathrm{f}}\,) = v$$

Read operation in the combined domain. To read a cell at address $((\mathbb{M}_1^\sharp : \alpha_1), \underline{\mathrm{f}})$ in a combined abstract memory $(m_1^\sharp, m_2^\sharp, i^\sharp)$, the analysis first attempts to read cell at address $(\alpha_1, \underline{\mathrm{f}})$ in $m_1^\sharp$. Therefore, the read operation derives from the following rule:

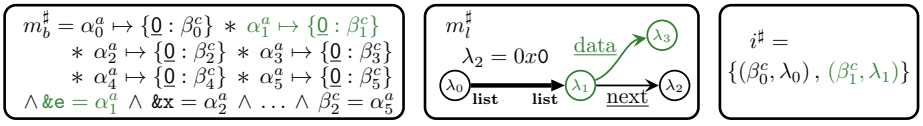
READDIRECT1
$$\frac{\mathit{read}_{\mathbb{M}_1^\sharp}(\alpha_1, \underline{\mathrm{f}}, m_1^\sharp) = \beta_1 + \underline{\mathrm{g}}}{\mathit{read}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}((\mathbb{M}_1^\sharp : \alpha_1), \underline{\mathrm{f}}, (m_1^\sharp, m_2^\sharp, i^\sharp)) = (\mathbb{M}_1^\sharp : \beta_1) + \underline{\mathrm{g}}}$$

The symmetric rule READDIRECT2 is shown in Appendix A.2. It may turn out that the cell at address $(\mathbb{M}_1^\sharp : \alpha_1)$ is abstracted in sub-memory $m_2^\sharp$ in which case rule READDIRECT1 cannot be applied. In fact, by the separation principle (Section 3.2), a cell is represented in exactly one sub-memory. To cope with this issue, the reading operation can retrieve the cell by looking for a symbolic name of sub-domain $\mathbb{M}_2^\sharp$ that is bound to α_1 by the abstract interface. In such cases, the definition of $\mathit{read}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}$ follows the rule:

READACROSS1
$$\frac{\mathit{read}_{\mathbb{M}_2^\sharp}(\alpha_2, \underline{\mathrm{f}}, m_2^\sharp) = \beta_2 + \underline{\mathrm{g}} \qquad i^\sharp \vdash \alpha_1 = \alpha_2}{\mathit{read}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}((\mathbb{M}_1^\sharp : \alpha_1), \underline{\mathrm{f}}, (m_1^\sharp, m_2^\sharp, i^\sharp)) = (\mathbb{M}_2^\sharp : \beta_2) + \underline{\mathrm{g}}}$$

The symmetric case is handled by rule READACROSS2 shown in Appendix A.2.

Example 4 (An evaluation across sub-memories). We consider the evaluation of the right hand side `e ->` $\underline{\mathrm{data}}$ of assignment at line 12 in Fig. 1 on the following combined abstract memory that is computed by the analysis after assignment at line 11 (assignment is treated in Section 4.3).



First, variable `e` is replaced by symbolic name $(\mathbb{M}_b^\sharp : \alpha_1^a)$ denoting its address and its content is evaluated to $(\mathbb{M}_b^\sharp : \beta_1^c)$. Then, reading cell at address $(\mathbb{M}_b^\sharp : \beta_1^c) + \underline{\mathrm{data}}$ fails in $\mathbb{M}_b^\sharp$ as the cell is actually abstracted in $\mathbb{M}_l^\sharp$. Therefore, the reading operation retrieves that cell at address $(\mathbb{M}_l^\sharp : \lambda_1) + \underline{\mathrm{data}}$, using the equality $(\beta_1^c, \lambda_1) \in i^\sharp$. Finally, the evaluation ends up with symbolic r-value $(\mathbb{M}_l^\sharp : \lambda_3)$.

4.3 Abstract transfer function for assignment

The analysis requires a set of abstract transfer functions handling operations such as assignment and test that need to evaluate l-value and r-value expressions [5]. Among those, the assignment is arguably the most sophisticated one, thus we describe only this operation here. The classical analysis of assignment $l = r$ shown in [5] proceeds as follows: (1) the left hand side is evaluated to a pair $\mathcal{L}_{\mathbb{M}^\sharp}[\![l]\!]\, m^\sharp = (\alpha, \underline{\mathrm{f}})$ representing the address of the cell that will be updated; (2) the right hand side is evaluated to a symbolic expression $\mathcal{R}_{\mathbb{M}^\sharp}[\![r]\!]\, m^\sharp = e$ representing the written value; and (3) the cell is updated in the abstract level, using the abstract *cell write* operation $\mathit{write}_{\mathbb{M}^\sharp}$ provided by the memory abstract domain $\mathbb{M}^\sharp$. Intuitively, $\mathit{write}_{\mathbb{M}^\sharp}(\alpha, \underline{\mathrm{f}}, e, m^\sharp)$ returns an abstract memory where the cell at address $\alpha + \underline{\mathrm{f}}$ has been updated to e. To state the soundness of this operation, we extend a valuation ν to cope with symbolic expressions in a natural way by defining $\overline{\nu}(\alpha + \underline{\mathrm{f}}) \stackrel{\text{def}}{=} \nu(\alpha) + \underline{\mathrm{f}}$ and $\overline{\nu}(v) \stackrel{\text{def}}{=} v$. Therefore, $\mathit{write}_{\mathbb{M}^\sharp}$ satisfies the condition:

$$(m, \nu) \in \gamma_{\mathbb{M}^\sharp}(m^\sharp) \;\Rightarrow\; \exists \nu' \supseteq \nu.\, (m[\nu(\alpha)+\underline{\mathrm{f}} \leftarrow \overline{\nu}(e)], \nu') \in \gamma_{\mathbb{M}^\sharp}(\, \mathit{write}_{\mathbb{M}^\sharp}(\alpha, \underline{\mathrm{f}}, e, m^\sharp)\,)$$

Cell write operation in a combined domain. A simple case occurs when left and right hand sides are both evaluated in the *same* sub-memory, in which case the cell write operation simply lifts computation to the corresponding sub-domain. However, a trickier case occurs when l-value and r-value are evaluated to *different* sub-memories, such as $((\mathbb{M}_1^\sharp : \alpha_1), \underline{\mathrm{f}})$ and $(\mathbb{M}_2^\sharp : \beta_2) + \underline{\mathrm{g}}$. In this case, the cell writing is performed in $m_1^\sharp$ as the cell requiring update is abstracted there. However, to avoid losing precision, the analysis needs a symbolic expression in $m_1^\sharp$ to relate the new content. Therefore, two cases may be encountered:

– β_2 is bound to a symbolic name $\beta_1 \in \mathcal{N}_{\mathbb{M}_1^\sharp}$ by the abstract interface, in which case $\mathit{write}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}$ is defined following the rule:

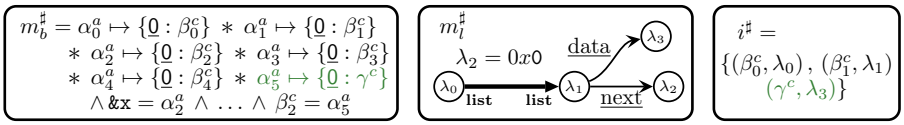
$$\textsc{WriteAcross1}\quad \frac{\mathit{write}_{\mathbb{M}_1^\sharp}(\alpha_1, \underline{\mathrm{f}}, \beta_1 + \underline{\mathrm{g}}, m_1^\sharp) = m_{1,\mathrm{post}}^\sharp \qquad i^\sharp \vdash \beta_1 = \beta_2}{\mathit{write}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}((\mathbb{M}_1^\sharp : \alpha_1), \underline{\mathrm{f}}, (\mathbb{M}_2^\sharp : \beta_2) + \underline{\mathrm{g}}, (m_1^\sharp, m_2^\sharp, i^\sharp)) = (m_{1,\mathrm{post}}^\sharp, m_2^\sharp, i^\sharp)}$$

– β_2 is not bound in the abstract interface, in which case a *fresh* variable β_1 is used to account for it in $m_1^\sharp$. Then $\mathit{write}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}$ is defined following the rule:

$$\textsc{WriteAcrossWeak1}\quad \frac{\mathit{write}_{\mathbb{M}_1^\sharp}(\alpha_1, \underline{\mathrm{f}}, \beta_1 + \underline{\mathrm{g}}, m_1^\sharp) = m_{1,\mathrm{post}}^\sharp \qquad \beta_1 \textit{ fresh in } m_1^\sharp}{\mathit{write}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}((\mathbb{M}_1^\sharp : \alpha_1), \underline{\mathrm{f}}, (\mathbb{M}_2^\sharp : \beta_2) + \underline{\mathrm{g}}, (m_1^\sharp, m_2^\sharp, i^\sharp)) = (m_{1,\mathrm{post}}^\sharp, m_2^\sharp, i^\sharp \cup \{(\beta_1, \beta_2)\})}$$

Example 5 (Assignment across sub-memories). We consider the computation of the post-condition of assignment `*x = e ->` $\underline{\mathrm{data}}$ at line 12 in Fig. 1, from the pre-condition shown in Example 4. The left and right hand sides are respectively evaluated to $((\mathbb{M}_b^\sharp : \alpha_5^a), \underline{0})$ and $(\mathbb{M}_l^\sharp : \lambda_3)$ (as shown in Example 4). Moreover, there is no symbolic name in $m_b^\sharp$ bound to λ_3 in $m_l^\sharp$ by the abstract interface. Therefore, $\mathit{write}_{\mathbb{M}_b^\sharp \circledast \mathbb{M}_l^\sharp}$

derives from rule WRITEACROSSWEAK1, and produces the following post-condition:



4.4 Inclusion checking

Fix-point computations [6] require widening and inclusion checking operators. In this section, we only detail the algorithm for inclusion checking as the widening algorithm is similar [5]. At a memory abstract domain $\mathbb{M}^\sharp$ level, the inclusion checking relies on the *abstract comparison* operator $\mathit{compare}_{\mathbb{M}^\sharp}$ that inputs two abstract memories $m_\mathrm{l}^\sharp$ and $m_\mathrm{r}^\sharp$ and returns a mapping Φ when it successfully establishes the abstract inclusion $m_\mathrm{l}^\sharp \sqsubseteq m_\mathrm{r}^\sharp$. Intuitively, the returned mapping relates symbolic names in $m_r^\sharp$ to symbolic names in $m_l^\sharp$ that valuations should map to the same value for the inclusion to hold. More formally, the soundness condition states the following:

$$\mathit{compare}_{\mathbb{M}^\sharp}(m_\mathrm{l}^\sharp, m_\mathrm{r}^\sharp) = \Phi \ \wedge\ (m, \nu) \in \gamma_{\mathbb{M}^\sharp}(m_\mathrm{l}^\sharp) \implies (m, \nu \circ \Phi) \in \gamma_{\mathbb{M}^\sharp}(m_\mathrm{r}^\sharp)$$

Inclusion checking in a combined domain. To compare the two combined abstract memories $m_l^\sharp = (m_{1,\mathrm{l}}^\sharp, m_{2,\mathrm{l}}^\sharp, i_l^\sharp)$ and $m_r^\sharp = (m_{1,\mathrm{r}}^\sharp, m_{2,\mathrm{r}}^\sharp, i_r^\sharp)$, the analysis first invokes the abstract comparisons of the sub-domains respectively on $(m_{1,\mathrm{l}}^\sharp, m_{1,\mathrm{r}}^\sharp)$ and $(m_{2,\mathrm{l}}^\sharp, m_{2,\mathrm{r}}^\sharp)$. When both succeed and thus return Φ_1 and Φ_2, the analysis checks the inclusion of the abstract interfaces by: $i_l^\sharp \sqsubseteq_{\Phi_1}^{\Phi_2} i_r^\sharp \iff \forall(\alpha_1, \alpha_2) \in i_r^\sharp.\ \ i_l^\sharp \vdash \Phi_1(\alpha_1) = \Phi_2(\alpha_2)$. Therefore, the abstract comparison operator is defined by the following rule:

$$\textsc{Incl}\ \frac{\mathit{compare}_{\mathbb{M}_1^\sharp}(m_{1,\mathrm{l}}^\sharp, m_{1,\mathrm{r}}^\sharp) = \Phi_1 \quad \mathit{compare}_{\mathbb{M}_2^\sharp}(m_{2,\mathrm{l}}^\sharp, m_{2,\mathrm{r}}^\sharp) = \Phi_2 \quad i_l^\sharp \sqsubseteq_{\Phi_1}^{\Phi_2} i_r^\sharp}{\mathit{compare}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}((m_{1,\mathrm{l}}^\sharp, m_{2,\mathrm{l}}^\sharp, i_l^\sharp), (m_{1,\mathrm{r}}^\sharp, m_{2,\mathrm{r}}^\sharp, i_r^\sharp)) = \Phi_1 \oplus \Phi_2}$$

Refinement using initial mappings. While *sound*, such a definition could lead to a loss of precision. For some memory abstract domains (such as the separating shape graphs domain), the abstract comparison operator internally initializes a mapping between symbolic names representing addresses of the same program variable (that valuations should clearly map to the same value). However, in a combined domain, a sub-memory with no such symbolic names is plausible (e.g. consider $m_l^\sharp$ in the combined abstract memory of Fig. 6(a)), and the sub-domain abstract comparison will therefore likely fail to establish the inclusion. To cope with that issue, the analysis provides an *initial mapping* as hint to the unsuccessful abstract comparison, that derives from the relationship inferred by the successful abstract comparison. More precisely, if $\mathit{compare}_{\mathbb{M}_1^\sharp}$ succeeds and returns Φ_1, the initial mapping defined by $\Phi_2^{\mathrm{init}}(\beta_{2,r}) = \beta_{2,l} \Leftrightarrow \exists \beta_{1,r} \in \mathcal{N}_{\mathbb{M}_1^\sharp}.\ i_r^\sharp \vdash \beta_{1,r} = \beta_{2,r} \ \wedge\ i_l^\sharp \vdash \Phi_1(\beta_{1,r}) = \beta_{2,l}$ can be passed as optional argument to $\mathit{compare}_{\mathbb{M}_2^\sharp}$.

Filename	MAD	#P	$\vee$	t(s)	%	tCF(s)	tSD(s)	#R	#RA
insert_remove.c	I<list>	3	2.09	0.248	basis	-	0.174	230	-
(158 LOC)	B $\circledast$ I<list>	3	2.09	0.151	60	0.035	0.055	230	16
balancing.c	I<tree>	3	2.56	0.501	basis	-	0.376	314	-
(188 LOC)	B $\circledast$ I<tree>	3	2.56	0.323	64	0.068	0.125	314	72
search_list_tree.c	I<list,tree>	5	3.40	0.330	basis	-	0.286	172	-
(138 LOC)	I<list> $\circledast$ I<tree>	5	3.40	0.364	110	0.031	0.292	172	48
	B $\circledast$ I<list,tree>	5	3.40	0.194	59	0.035	0.098	172	70
	B $\circledast$ I<list> $\circledast$ I<tree>	5	3.40	0.231	70	0.071	0.113	172	70

Fig. 8. Analysis results (measured on a 2.2 Ghz Intel Core i7 with 8 GB of RAM): MAD is the memory abstract domain used (B stands for the *bounded data structure* domain, I<.> stands for the *separating shape graphs* domain instantiated with inductive definitions that are either list or tree, $\circledast$ stands for separating combination of domains), #P is the number of properties proven by the analysis, $\vee$ is average number of disjuncts at each program point, t is the total analysis time in seconds,% is the time of analysis compared to analysis using a monolithic domain, tCF (resp. tSD) is the time of analysis spent in the combination functor (resp. sub-domains), #R is the number of read operation calls and #RA is the number of read operations crossing sub-memories.

5 Implementation and empirical evaluation

We test empirically the precision and efficiency of the combined analysis compared to a monolithic one and describe the results here. The separating combination described in this paper is implemented in the MemCAD analyzer[3]. The analysis is fully automatic. It takes as input C code and the desired structure of the memory abstract domain. The two analysis variants were applied to a set of over 15 micro-benchmarks, similar to the code fragment in Fig. 1. We verify memory safety properties, such as the absence of null pointer dereferences, as well as structural assertions (annotated in the code). In Fig. 8, we report on some representative analysis results relevant to questions in this paper. The C programs considered consist of data structure-manipulation routines (e.g. insertion, deletion, search) for lists and trees either called sequentially or interleaved. They can all be analyzed using a monolithic domain.

First, we note that importantly the combined analyses retain the same level of precision as the monolithic analyses in terms of the number of properties that can be proven (column #P). For each program, the number of properties proven on the first line (monolithic) is the same as the number proven on the subsequent lines (various combinations). The key part of the combined analysis is the interface between sub-memories. Its necessity is demonstrated by the ratio of read operations that *cross* the sub-memories in the combined analyses (column #RA over #R).

Next, we consider the relative efficiency of the various memory abstract domain combinations with respect to the monolithic version. Regardless of configuration, the MemCAD analyzer computes for each program point, a finite disjunction of abstract memories. We first observe that the use of a combined domain does not introduce an extra combinatorial factor as the number of disjuncts is the same for the monolithic and the combined analyses (column $\vee$). To probe into the overhead of our combination functor, we considered in search_list_tree.c decomposing the memory abstract domain into list- and tree-specific regions (I<list> $\circledast$ I<tree>). In this case, the list and tree do-

[3] `http://www.di.ens.fr/~rival/memcad.html`

mains are instantiations of the same generic, parametric separating shape graph domain. Thus, this instantiation pays for the overhead of the separating combination without the benefit of an optimized sub-domain. We observe that there is an overhead, but it seems acceptable given that the separating combination offers the possibility of replacing the sub-domains with specialized and optimized versions (a ratio of 110%-120% in the two instances shown here).

The win with our separating combination functor comes from applying it with an optimized sub-domain. In the variants with $\mathrm{B} \circledast \cdots$, we use a *bounded data structure* domain to efficiently manage the bounded part of the memory (e.g. the top activation record in the call stack). This sub-domain is implemented efficiently knowing that it only needs to abstract bounded data structures. From Fig. 8, we see that separating out the bounded part of memory into a more efficient specialized domain is highly effective—noticeably decreasing the overall analysis times despite the overhead of combination (a ratio of around 60% in all cases).

6 Related work

The first important abstract domain combination operation to be introduced is the reduced product [7], which has enabled constructing very expressive abstract domains from simpler ones. Intuitively, a property is decomposed into a conjunction of (possibly radically different) basic properties. This construction was applied to a wide range of analyzers, including ASTRÉE [3], where a large set of numerical abstract domains exchange information over a chain of reduced products [8]. The benefit of reduced product is even greater for libraries of abstractions with a common interface such as APRON [13]. It was also used to describe the Nelson-Oppen procedure [9].

Our contribution seeks to simplify abstract domain construction, while allowing greater expressiveness. It exploits *separation* [17], albeit in a different way than the numerous shape analyses that exploit it in the definition of their summarization predicates [10,2,4]. In these analyses, separation permits (hopefully all) updates to be handled as strong updates, which is crucial for both precision and efficiency. Our analysis exploits separation so as to combine independent memory abstract domains, so as to achieve at least the same precision and better efficiency by delegating the abstraction of particular data structures to the most appropriate sub-domains. Note that the sub-domains may (and in all the examples shown in this paper, do) also make use of separation as the aforementioned analyses. In [23], separation was used to represent distinct heap regions using heterogeneous abstractions, yet this work relies on code specifications transformed into sub-problems handled by different abstractions, and proceeds by verification, although our combinator allows inference of invariants.

Other combinations of abstractions have been proposed so as to enhance memory analyses. In particular, [14] presents an approach that uses classical conjunction together with zone variables to relate corresponding regions. Moreover, [11] combines formulae by distinguishing per-field and per-object separating conjunctions. In [15], sets of sub-graphs are used to represent properties about non-correlated data structures and to realize a gain in performance. These analyses are based on problem specific de-

compositions while our domain combinator is generic, in the sense that it does not make any assumptions on the way the memory properties are represented in the sub-domains.

In previous work [22], we proposed a reduced product for memory abstractions as a generic abstract domain combinator. This combinator does not rely on separation and provides a different form of separation of concerns than our separating combinator: in [22], sub-domains express a collection of properties of the same structure whereas the separating conjunction operator combines domains representing distinct structures. Moreover, we introduced a hierarchical memory abstraction to abstract structures allocated inside other structures [21]; in that work the whole memory is abstracted in the main domain, and a sub-domain describes nested structures. These combinators are implemented as ML functors in the MemCAD analyzer and can be used together (although assessing such compositions is beyond the scope of this paper).

7 Conclusion

In this paper, we introduced a combinator for separately conjoining memory abstract domains, enabling composite analyses that are precise, efficient, and flexible. Our proposal enables a separation of concerns when designing static analyses that need to deal with complex data structures, as very different domains can be combined to abstract disjoint memory regions. A natural extension of our study would be to integrate other memory abstractions, as found in 3-valued logic shape analyses [19,1,12], into our framework.

References

1. G. Arnold, R. Manevich, M. Sagiv, and R. Shaham. Combining shape analyses by intersecting abstractions. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2006.
2. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *Conference on Computer Aided Verification (CAV)*, 2007.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Programming Languages Design and Implementation (PLDI)*, 2003.
4. B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *Principles Of Programming Languages (POPL)*, 2008.
5. B.-Y. E. Chang and X. Rival. Modular construction of shape-numeric analyzers. In *SAIRP*, 2013.
6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles Of Programming Languages (POPL)*, 1977.
7. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Principles Of Programming Languages (POPL)*, 1979.
8. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the astrée static analyzer. In *Advances in Computer Science (ASIAN)*, 2006.

9. P. Cousot, R. Cousot, and L. Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In *Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, 2011.
10. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2006.
11. C. Dragoi, C. Enea, and M. Sighireanu. Local shape analysis for overlaid data structures. In *Static Analysis Symposium (SAS)*, 2013.
12. P. Ferrara, R. Fuchs, and U. Juhasz. TVLA+ : TVLA and value analyses together. In *Software Engineering and Formal Methods (SEFM)*, 2012.
13. B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Conference on Computer Aided Verification (CAV)*, 2009.
14. O. Lee, H. Yang, and R. Petersen. Program analysis for overlaid data structures. In *Conference on Computer Aided Verification (CAV)*, 2011.
15. R. Manevich, J. Berdine, B. Cook, G. Ramalingam, and M. Sagiv. Shape analysis by graph decomposition. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2007.
16. A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2006.
17. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logic in Computer Science (LICS)*, 2002.
18. X. Rival and B.-Y. E. Chang. Calling context abstraction with shapes. In *Principles Of Programming Languages (POPL)*, 2011.
19. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Principles Of Programming Languages (POPL)*, 1999.
20. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages And Systems (TOPLAS)*, 2002.
21. P. Sotin and X. Rival. Hierarchical shape abstraction of dynamic structures in static blocks. In *Asian Conference on Programming Languages and Software (APLAS)*, 2012.
22. A. Toubhans, B.-Y. E. Chang, and X. Rival. Reduced product combination of abstract domains for shapes. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2013.
23. E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous asbtractions. In *Programming Languages Design and Implementation (PLDI)*, 2004.

A Rules for abstract transfer functions of the combined domains

In this section, we give the rules for defining abstract transfer functions of a combined memory abstract domain $\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp$, from the abstract transfer functions of memory abstract sub-domains $\mathbb{M}_1^\sharp$ and $\mathbb{M}_2^\sharp$.

A.1 Cell creation operation

$$\boxed{new_{\mathbb{M}^\sharp} : \texttt{int} \times \mathbb{M}^\sharp \to \mathcal{N}_{\mathbb{M}^\sharp} \times \mathbb{M}^\sharp}$$

In a combined domain, the cell creation should be handled by exactly one domain (to avoid violating the separation principle). Therefore, depending on which submemory is chosen to account for the new cell, we give two definitions for the cell

creation operation, that derives from the following rules:

NEW1
$$\frac{\mathit{new}_{\mathbb{M}_1^\sharp}(\mathtt{s}, m_1^\sharp) = (\alpha_1, m_{1,\mathrm{post}}^\sharp)}{\mathit{new}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}(\mathtt{s}, (m_1^\sharp, m_2^\sharp, i^\sharp)) = ((\mathbb{M}_1^\sharp : \alpha_1), (m_{1,\mathrm{post}}^\sharp, m_2^\sharp, i^\sharp))}$$

NEW2
$$\frac{\mathit{new}_{\mathbb{M}_2^\sharp}(\mathtt{s}, m_2^\sharp) = (\alpha_2, m_{2,\mathrm{post}}^\sharp)}{\mathit{new}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}(\mathtt{s}, (m_1^\sharp, m_2^\sharp, i^\sharp)) = ((\mathbb{M}_2^\sharp : \alpha_2), (m_1^\sharp, m_{2,\mathrm{post}}^\sharp, i^\sharp))}$$

A.2 Read operation

$$\boxed{\mathit{read}_{\mathbb{M}^\sharp} : \mathcal{N}_{\mathbb{M}^\sharp} \times \mathbb{F} \times \mathbb{M}^\sharp \rightharpoonup \mathcal{E}_{\mathbb{M}^\sharp}}$$

The read operation (Section 4.2) extracts the content of a cell in the abstract level. In a combined domain, cell read operation first attempts to extract the content of a cell in the sub-memory where the symbolic name representing its address belongs. Therefore, the cell read operation is first defined by the two following rules:

READDIRECT1
$$\frac{\mathit{read}_{\mathbb{M}_1^\sharp}(\alpha_1, \underline{\mathrm{f}}, m_1^\sharp) = \beta_1 + \underline{\mathrm{g}}}{\mathit{read}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}((\mathbb{M}_1^\sharp : \alpha_1), \underline{\mathrm{f}}, (m_1^\sharp, m_2^\sharp, i^\sharp)) = (\mathbb{M}_1^\sharp : \beta_1) + \underline{\mathrm{g}}}$$

READDIRECT2
$$\frac{\mathit{read}_{\mathbb{M}_2^\sharp}(\alpha_2, \underline{\mathrm{f}}, m_2^\sharp) = \beta_2 + \underline{\mathrm{g}}}{\mathit{read}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}((\mathbb{M}_2^\sharp : \alpha_2), \underline{\mathrm{f}}, (m_1^\sharp, m_2^\sharp, i^\sharp)) = (\mathbb{M}_2^\sharp : \beta_2) + \underline{\mathrm{g}}}$$

However, cell read operation of sub-domains may fail. In such cases, the cell read operation attempts to retrieve the cell using the abstract interface. It is then defined by the two rules:

READACROSS1
$$\frac{\mathit{read}_{\mathbb{M}_2^\sharp}(\alpha_2, \underline{\mathrm{f}}, m_2^\sharp) = \beta_2 + \underline{\mathrm{g}} \qquad i^\sharp \vdash \alpha_1 = \alpha_2}{\mathit{read}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}((\mathbb{M}_1^\sharp : \alpha_1), \underline{\mathrm{f}}, (m_1^\sharp, m_2^\sharp, i^\sharp)) = (\mathbb{M}_2^\sharp : \beta_2) + \underline{\mathrm{g}}}$$

READACROSS2
$$\frac{\mathit{read}_{\mathbb{M}_1^\sharp}(\alpha_1, \underline{\mathrm{f}}, m_1^\sharp) = \beta_1 + \underline{\mathrm{g}} \qquad i^\sharp \vdash \alpha_1 = \alpha_2}{\mathit{read}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}((\mathbb{M}_2^\sharp : \alpha_2), \underline{\mathrm{f}}, (m_1^\sharp, m_2^\sharp, i^\sharp)) = (\mathbb{M}_1^\sharp : \beta_1) + \underline{\mathrm{g}}}$$

A.3 Cell write operation

$$\boxed{\mathit{write}_{\mathbb{M}^\sharp} : \mathcal{N}_{\mathbb{M}^\sharp} \times \mathbb{F} \times \mathcal{E}_{\mathbb{M}^\sharp} \times \mathbb{M}^\sharp \to \mathbb{M}^\sharp}$$

The analysis uses the cell write operation (Section 4.3) to compute post-condition of assignments. It updates the content of a cell in the abstract level. In a combined domain,

several cases may be encountered depending on which sub-memory belong the address of the updated cell and the written content. When both are in the same sub-memory, the cell writing operation simply lifts the sub-domain operation:

WRITEDIRECT1

$$\frac{\mathit{write}_{\mathbb{M}_1^\sharp}(\alpha_1, \underline{\mathrm{f}}, \beta_1 + \underline{\mathrm{g}}, m_1^\sharp) = m_{1,\mathrm{post}}^\sharp}{\mathit{write}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}((\mathbb{M}_1^\sharp : \alpha_1), \underline{\mathrm{f}}, (\mathbb{M}_1^\sharp : \beta_1) + \underline{\mathrm{g}}, (m_1^\sharp, m_2^\sharp, i^\sharp)) = (m_{1,\mathrm{post}}^\sharp, m_2^\sharp, i^\sharp)}$$

WRITEDIRECT2

$$\frac{\mathit{write}_{\mathbb{M}_2^\sharp}(\alpha_2, \underline{\mathrm{f}}, \beta_2 + \underline{\mathrm{g}}, m_1^\sharp) = m_{2,\mathrm{post}}^\sharp}{\mathit{write}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}((\mathbb{M}_2^\sharp : \alpha_2), \underline{\mathrm{f}}, (\mathbb{M}_2^\sharp : \beta_2) + \underline{\mathrm{g}}, (m_1^\sharp, m_2^\sharp, i^\sharp)) = (m_1^\sharp, m_{2,\mathrm{post}}^\sharp, i^\sharp)}$$

When the written content is a value v, the cell reading operation is defined by the two following rules:

WRITEVAL1

$$\frac{\mathit{write}_{\mathbb{M}_1^\sharp}(\alpha_1, \underline{\mathrm{f}}, v, m_1^\sharp) = m_{1,\mathrm{post}}^\sharp}{\mathit{write}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}((\mathbb{M}_1^\sharp : \alpha_1), \underline{\mathrm{f}}, v, (m_1^\sharp, m_2^\sharp, i^\sharp)) = (m_{1,\mathrm{post}}^\sharp, m_2^\sharp, i^\sharp)}$$

WRITEVAL2

$$\frac{\mathit{write}_{\mathbb{M}_2^\sharp}(\alpha_2, \underline{\mathrm{f}}, v, m_1^\sharp) = m_{2,\mathrm{post}}^\sharp}{\mathit{write}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}((\mathbb{M}_2^\sharp : \alpha_2), \underline{\mathrm{f}}, v, (m_1^\sharp, m_2^\sharp, i^\sharp)) = (m_1^\sharp, m_{2,\mathrm{post}}^\sharp, i^\sharp)}$$

The four following rules define read operation when the address of the updated cell and the written value are in different sub-memories.

WRITEACROSS1

$$\frac{\mathit{write}_{\mathbb{M}_1^\sharp}(\alpha_1, \underline{\mathrm{f}}, \beta_1 + \underline{\mathrm{g}}, m_1^\sharp) = m_{1,\mathrm{post}}^\sharp \qquad i^\sharp \vdash \beta_1 = \beta_2}{\mathit{write}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}((\mathbb{M}_1^\sharp : \alpha_1), \underline{\mathrm{f}}, (\mathbb{M}_2^\sharp : \beta_2) + \underline{\mathrm{g}}, (m_1^\sharp, m_2^\sharp, i^\sharp)) = (m_{1,\mathrm{post}}^\sharp, m_2^\sharp, i^\sharp)}$$

WRITEACROSS2

$$\frac{\mathit{write}_{\mathbb{M}_2^\sharp}(\alpha_2, \underline{\mathrm{f}}, \beta_2 + \underline{\mathrm{g}}, m_2^\sharp) = m_{2,\mathrm{post}}^\sharp \qquad i^\sharp \vdash \beta_1 = \beta_2}{\mathit{write}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}((\mathbb{M}_2^\sharp : \alpha_2), \underline{\mathrm{f}}, (\mathbb{M}_1^\sharp : \beta_1) + \underline{\mathrm{g}}, (m_1^\sharp, m_2^\sharp, i^\sharp)) = (m_1^\sharp, m_{2,\mathrm{post}}^\sharp, i^\sharp)}$$

WRITEACROSSWEAK1

$$\frac{\mathit{write}_{\mathbb{M}_1^\sharp}(\alpha_1, \underline{\mathrm{f}}, \beta_1 + \underline{\mathrm{g}}, m_1^\sharp) = m_{1,\mathrm{post}}^\sharp \qquad \beta_1 \textit{ fresh in } m_1^\sharp}{\mathit{write}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}((\mathbb{M}_1^\sharp : \alpha_1), \underline{\mathrm{f}}, (\mathbb{M}_2^\sharp : \beta_2) + \underline{\mathrm{g}}, (m_1^\sharp, m_2^\sharp, i^\sharp)) = (m_{1,\mathrm{post}}^\sharp, m_2^\sharp, i^\sharp \cup \{(\beta_1, \beta_2)\})}$$

WRITEACROSSWEAK2

$$\frac{\mathit{write}_{\mathbb{M}_2^\sharp}(\alpha_2, \underline{\mathrm{f}}, \beta_2 + \underline{\mathrm{g}}, m_1^\sharp) = m_{2,\mathrm{post}}^\sharp \qquad \beta_2 \textit{ fresh in } m_2^\sharp}{\mathit{write}_{\mathbb{M}_1^\sharp \circledast \mathbb{M}_2^\sharp}((\mathbb{M}_2^\sharp : \alpha_2), \underline{\mathrm{f}}, (\mathbb{M}_1^\sharp : \beta_1) + \underline{\mathrm{g}}, (m_1^\sharp, m_2^\sharp, i^\sharp)) = (m_1^\sharp, m_{2,\mathrm{post}}^\sharp, i^\sharp \cup \{(\beta_1, \beta_2)\})}$$